Imperfect-Information Games for System Design

Dietmar Berwanger[1] Laurent Doyen[2]

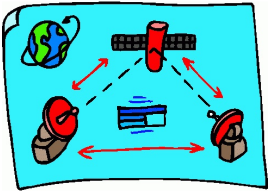
[1]LSV, CNRS & ENS Cachan

[2]Université Libre de Bruxelles

PSY Grenoble, June 2009

Games for Verification and Synthesis – in a Nutshell

Systems ▶ Models



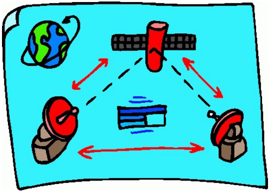
Games for Verification and Synthesis – in a Nutshell

Systems ▶ Models

security, dependability
concurrency, real-time
usability, . . .
non-terminating dynamics
modularity & interaction

Games for Verification and Synthesis – in a Nutshell

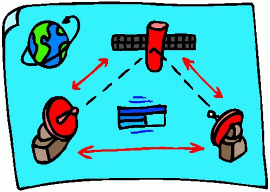
Systems ▶ Models



Specifications ▶ Logics

Games for Verification and Synthesis – in a Nutshell

Systems ▶ Models



Specifications ▶ Logics

avoid failure AG¬

ensure progress AGEF¬ψ

assume – guarantee $||\,\psi$

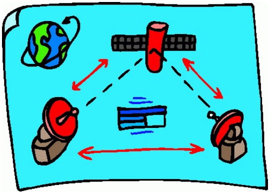
compositionality

interactive analysis

Games for Verification and Synthesis – in a Nutshell

Systems ▶ Models

Specifications ▶ Logics



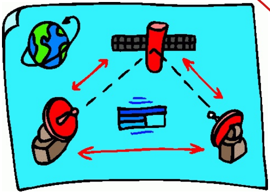
Games

- uniform framework
- modular and interactive

Games for Verification and Synthesis – in a Nutshell

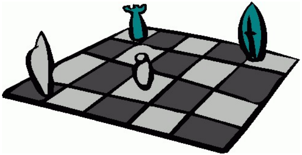
Systems ▶ Models

Specifications ▶ Logics



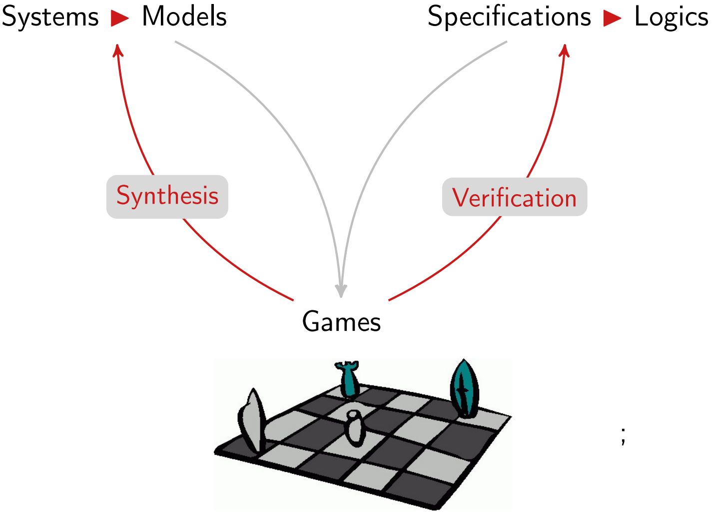
compile

Games



;

Games for Verification and Synthesis – in a Nutshell



;

Why imperfect information ?

Example

```
void main () {
     int got_lock = 0;
     do {
1:           if (*) {
2:                lock ();
3:                got_lock++;
             }
4:           if (got_lock != 0) {
5:                unlock ();
             }
6:           got_lock--;
     } while (*);
}
```

```
void lock () {
     assert(L == 0);
     L = 1;
}
```

```
void unlock () {
     assert(L == 1);
     L = 0;
}
```

Example

```
void main () {
     int got_lock = 0;
     do {
1:          if (*) {
2:               lock ();
3:               got_lock++;
            }
4:          if (got_lock != 0) {
5:               unlock ();
            }
6:          got_lock--;
     } while (*);
}
```

Wrong!

```
void lock () {
     assert(L == 0);
     L = 1;
}
```

```
void unlock () {
     assert(L == 1);
     L = 0;
}
```

Example

```
void main () {
    int got_lock = 0;
    do {
1:      if (*) {
2:          lock ();
3:          s0 | s1 | inc | dec;
        }
4:      if (got_lock != 0) {
5:          unlock ();
        }
6:      s0 | s1 | inc | dec;
    } while (*);
}
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

s0 ≡ got_lock = 0
s1 ≡ got_lock = 1
Inc ≡ got_lock++
dec ≡ got_lock--

Example

```
void main () {
    int got_lock = 0;
    do {
1:      if (*) {
2:          lock ();
3:          s0 | s1 | inc | dec;
        }
4:      if (got_lock != 0) {
5:          unlock ();
        }
6:      s0 | s1 | inc | dec;
    } while (*);
}
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

Repair/synthesis as a game:

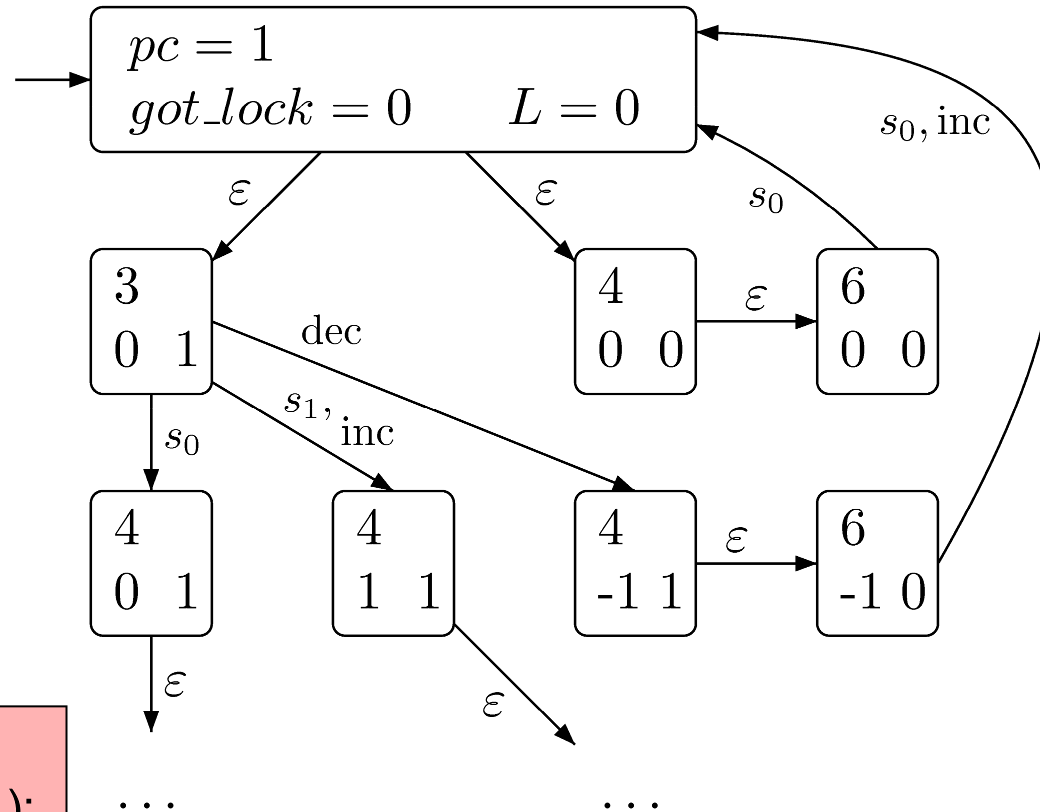
- System vs. Environment
- Turn-based game graph
- ω-regular objective

Example

```
void main () {
      int got_lock = 0;
      do {
1:          if (*) {
2:                lock ();
3:                s0 | s1 | inc | dec;
            }
4:          if (got_lock != 0) {
5:                unlock ();
            }
6:          s0 | s1 | inc | dec;
      } while (*);
}
```

```
void lock () {
      assert(L == 0);
      L = 1;
}
```

```
void unlock () {
      assert(L == 1);
      L = 0;
}
```

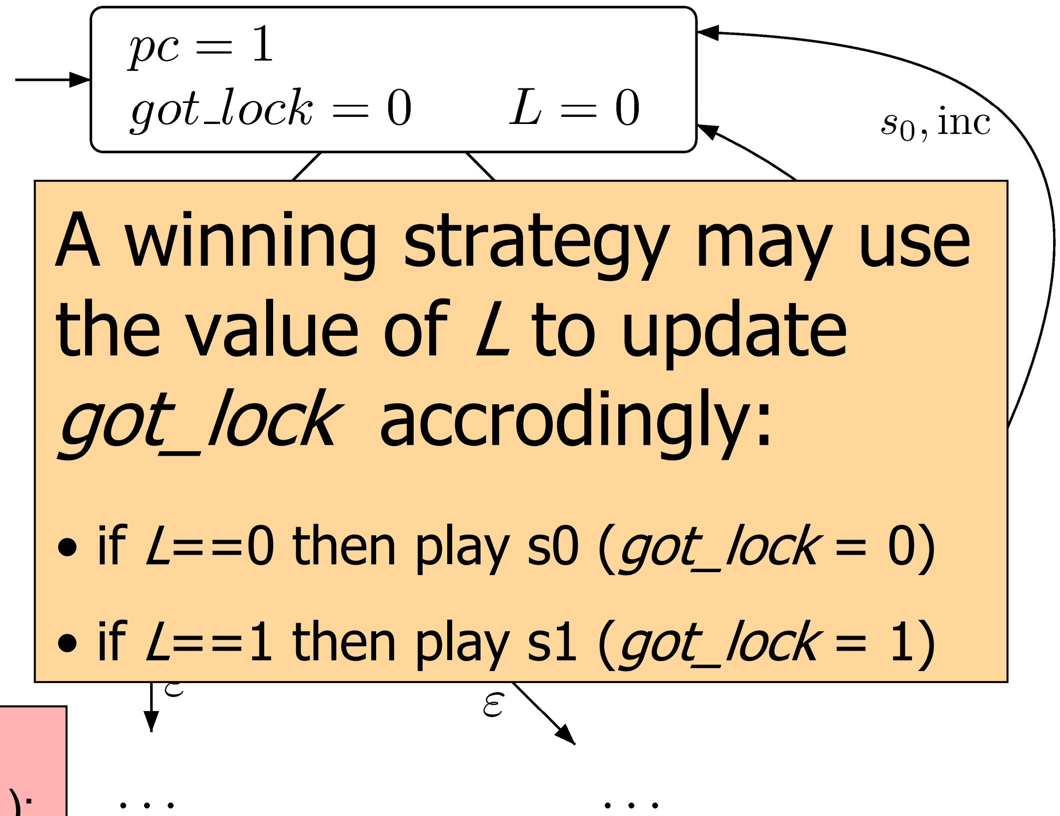


Example

```
void main () {
    int got_lock = 0;
    do {
1:      if (*) {
2:          lock ();
3:          s0 | s1 | inc | dec;
        }
4:      if (got_lock != 0) {
5:          unlock ();
        }
6:      s0 | s1 | inc | dec;
    } while (*);
}
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```



A winning strategy may use the value of *L* to update *got_lock* accrodingly:

- if *L*==0 then play s0 (*got_lock* = 0)
- if *L*==1 then play s1 (*got_lock* = 1)

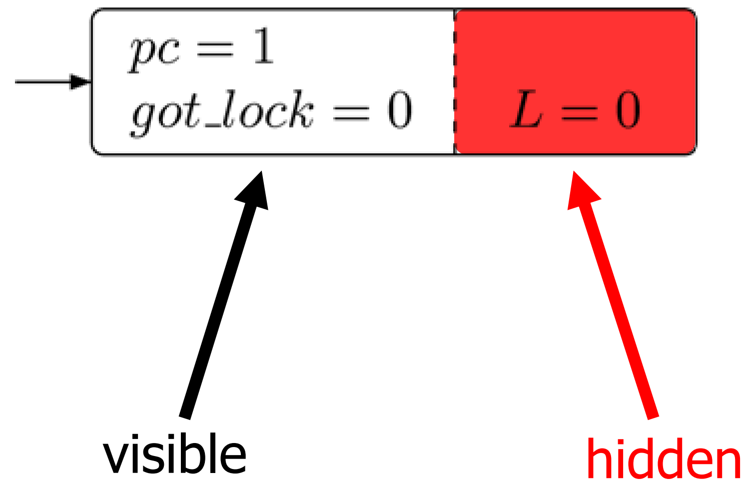
Example

```
void main () {
     int got_lock = 0;
     do {
1:          if (*) {
2:               lock ();
3:               s0 | s1 | inc | dec;
            }
4:          if (got_lock != 0) {
5:               unlock ();
            }
6:          s0 | s1 | inc | dec;
     } while (*);
}
```

```
void lock () {
     assert(L == 0);
     L = 1;
}
```

```
void unlock () {
     assert(L == 1);
     L = 0;
}
```

Repair/synthesis as a game of imperfect information:



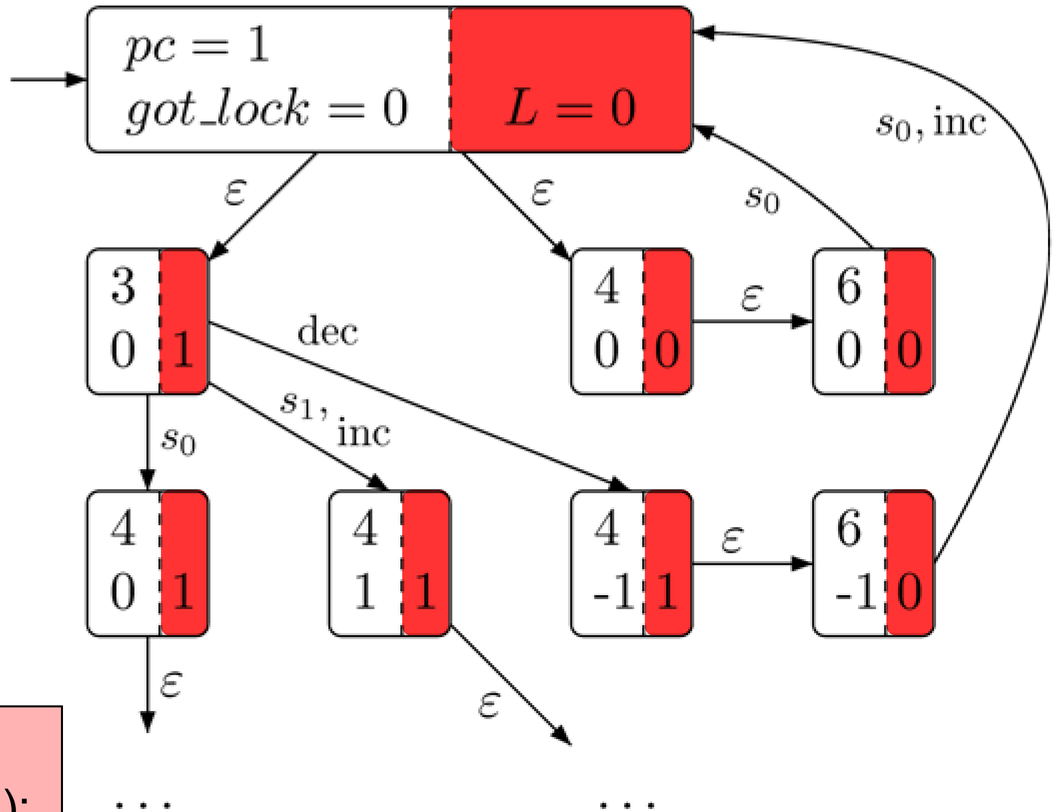
States that differ only by the value of *L* have the same observation

Example

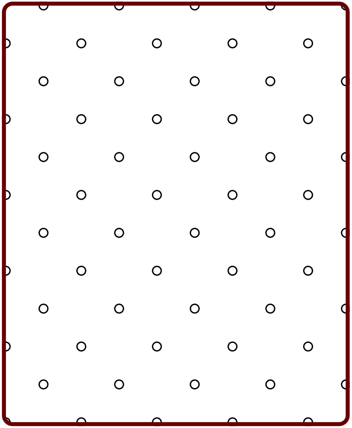
```
void main () {
    int got_lock = 0;
    do {
1:      if (*) {
2:          lock ();
3:          s0 | s1 | inc | dec;
        }
4:      if (got_lock != 0) {
5:          unlock ();
        }
6:      s0 | s1 | inc | dec;
    } while (*);
}
```

```
void lock () {
    assert(L == 0);
    L = 1;
}
```

```
void unlock () {
    assert(L == 1);
    L = 0;
}
```

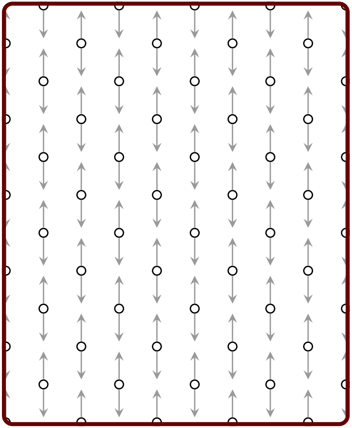


Transition structure with imperfect information



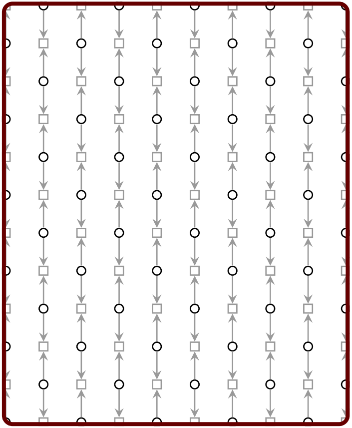
- States

Transition structure with imperfect information



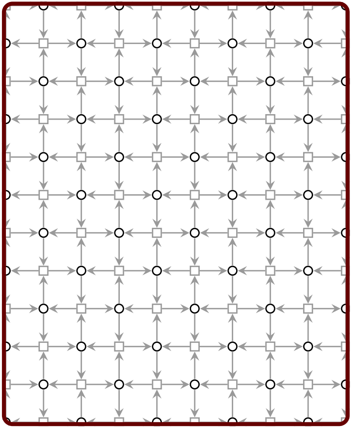
- States
- Player 1 – actions: $\uparrow, \downarrow$

Transition structure with imperfect information



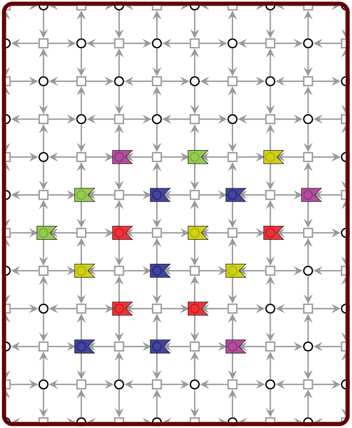
- States
- Player 1 – actions: ↑, ↓

Transition structure with imperfect information



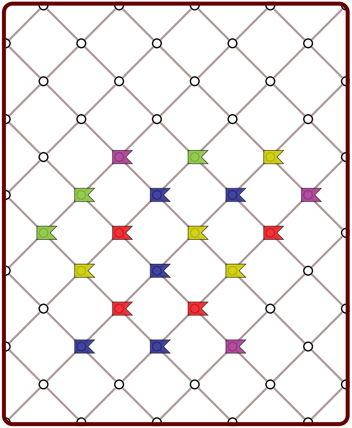
- States
- Player 1 – actions: $\uparrow, \downarrow$
- Transitions

Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

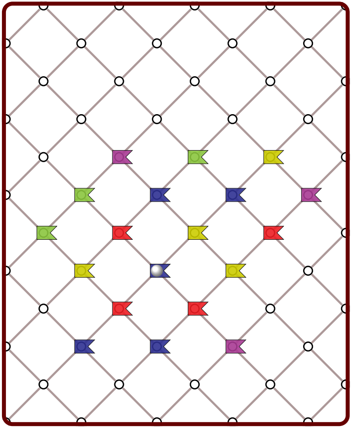
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play:

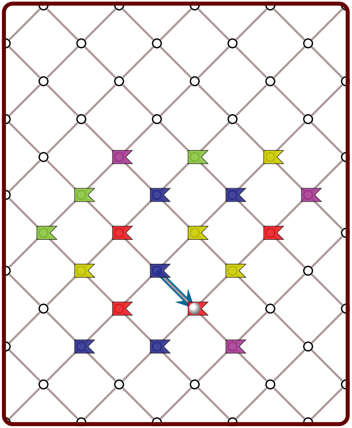
Transition structure with imperfect information



- States
- Player 1 – actions: $\uparrow, \downarrow$
- Transitions
- Player 2 – observations:

Play:

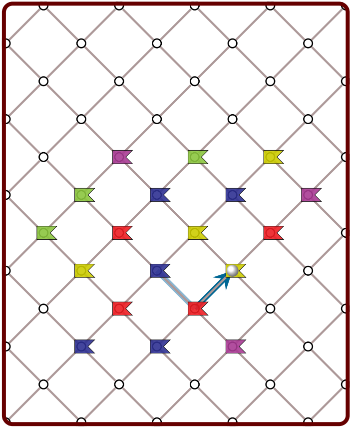
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓

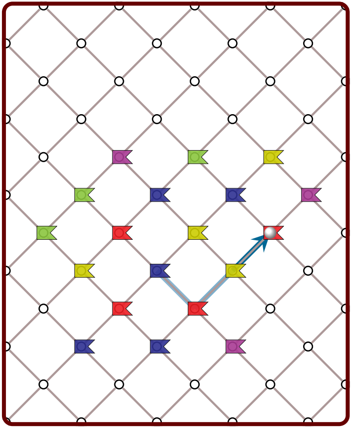
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑

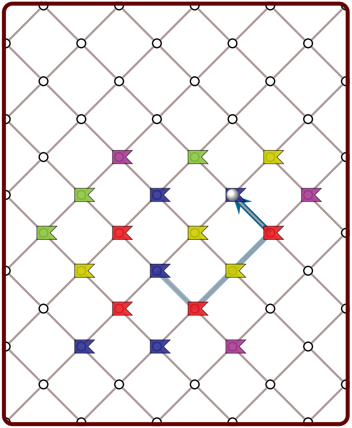
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑

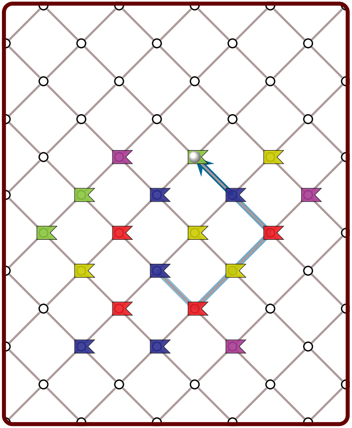
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑

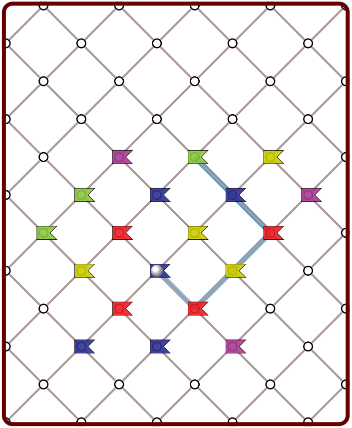
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑ ↑

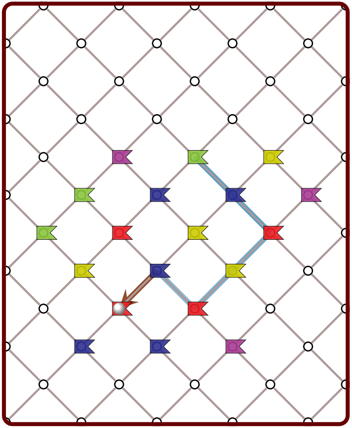
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑ ↑ ...

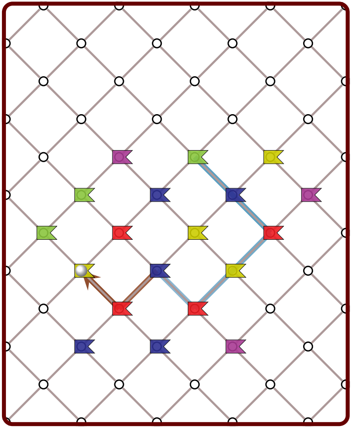
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑ ↑ . . .

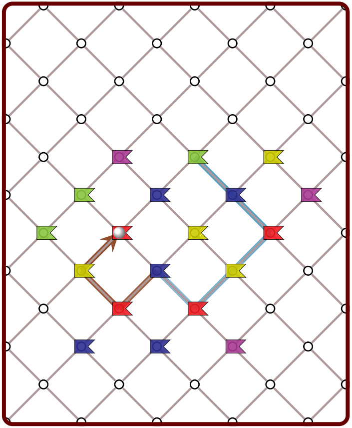
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑ ↑ . . .

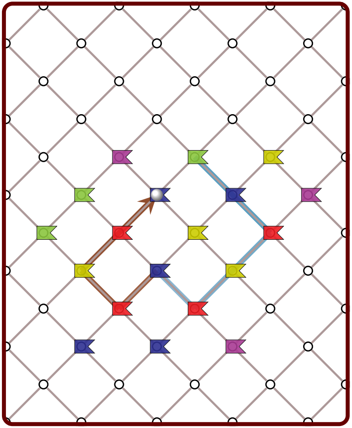
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑ ↑ . . .

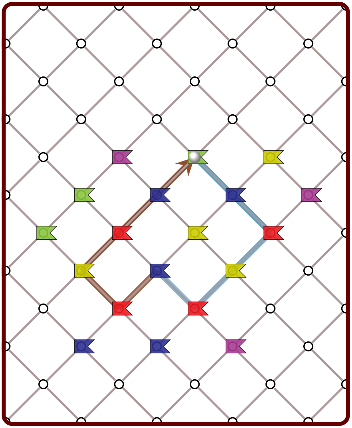
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play:

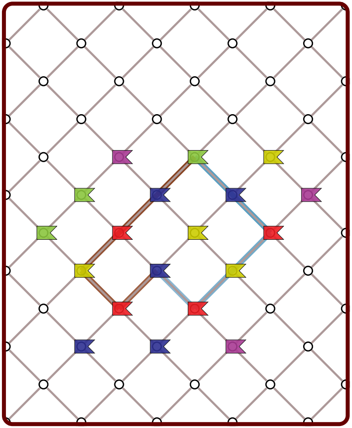
Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations:

Play: ↓ ↑ ↑ ↑ ↑ ...

Transition structure with imperfect information



- States
- Player 1 – actions: ↑, ↓
- Transitions
- Player 2 – observations: yellow, red, green, purple and blue flags

Play: blue flag ↓ red flag ↑ yellow flag ↑ red flag ↑ blue flag ↑ green flag . . .

Algorithms

Imperfect information

Games of imperfect information can be solved by a reduction to games of perfect information.

G,Obs → G' → Winning region

Imperfect information ↓ subset construction

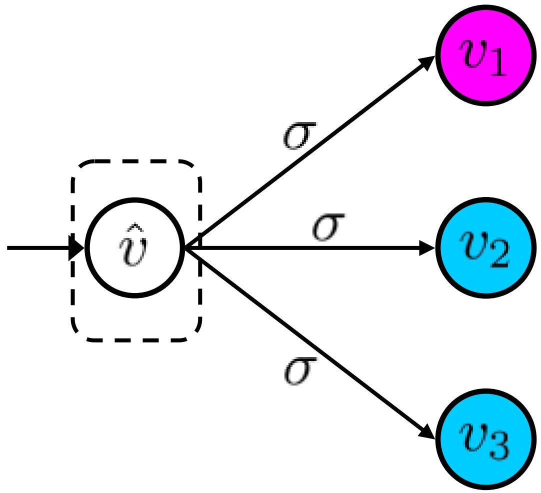
Perfect information ↓ classical techniques

Subset construction

After a finite prefix of a play, Player 1 has a partial knowledge of the current state of the game: **a set of states**, called a **cell**.

Subset construction

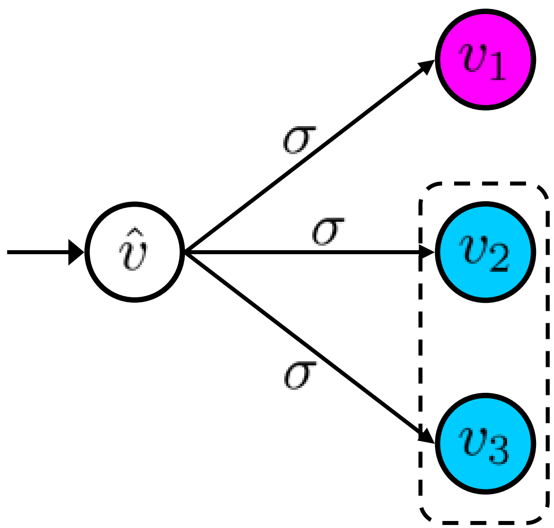
After a finite prefix of a play, Player 1 has a partial knowledge of the current state of the game: **a set of states**, called a **cell**.



Initial knowledge: cell $\{\widehat{v}\}$

Subset construction

After a finite prefix of a play, Player 1 has a partial knowledge of the current state of the game: **a set of states**, called a **cell**.



Initial knowledge: cell $\{\hat{v}\}$

Player 1 plays σ,

Player 2 chooses v_2.

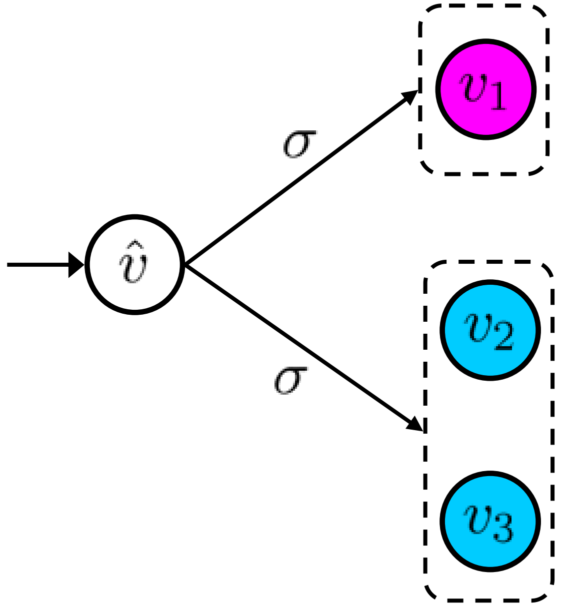
Current knowledge: cell $\{v_2, v_3\}$

$\downarrow$

$\mathsf{Post}_\sigma(\{\hat{v}\}) \cap o_2$

Subset construction

After a finite prefix of a play, Player 1 has a partial knowledge of the current state of the game: **a set of states**, called a **cell**.



Subset construction [Reif84]:

- keeps track of the knowledge
- yields equivalent game of perfect information with macro-states (=cell)

Classical solution

Powerset construction [Reif84]:

- keeps track of the knowledge of System
- yields equivalent game of perfect information

Memoryless strategies (in perfect-information) translate to finite-memory strategies

(memory automaton tracks set of possible positions)

Complexity

- Problem is EXPTIME-complete
  (even for safety and reachability)
- Exponential memory might be needed

The powerset solution [Reif84]

- is an exponential construction
- is not on-the-fly
- is independent of the objective

Can we do better ?

Imperfect information

G,Obs → G′ → Winning region

Imperfect information ↓ subset construction

Exponential blow-up

Perfect information ↓ classical techniques

Imperfect information

G,Obs → G′ → Winning region

implicit

Imperfect information

Perfect information

Direct symbolic algorithm

Imperfect information

G,Obs ➔ G′ ➔ Winning region

implicit

Imperfect information

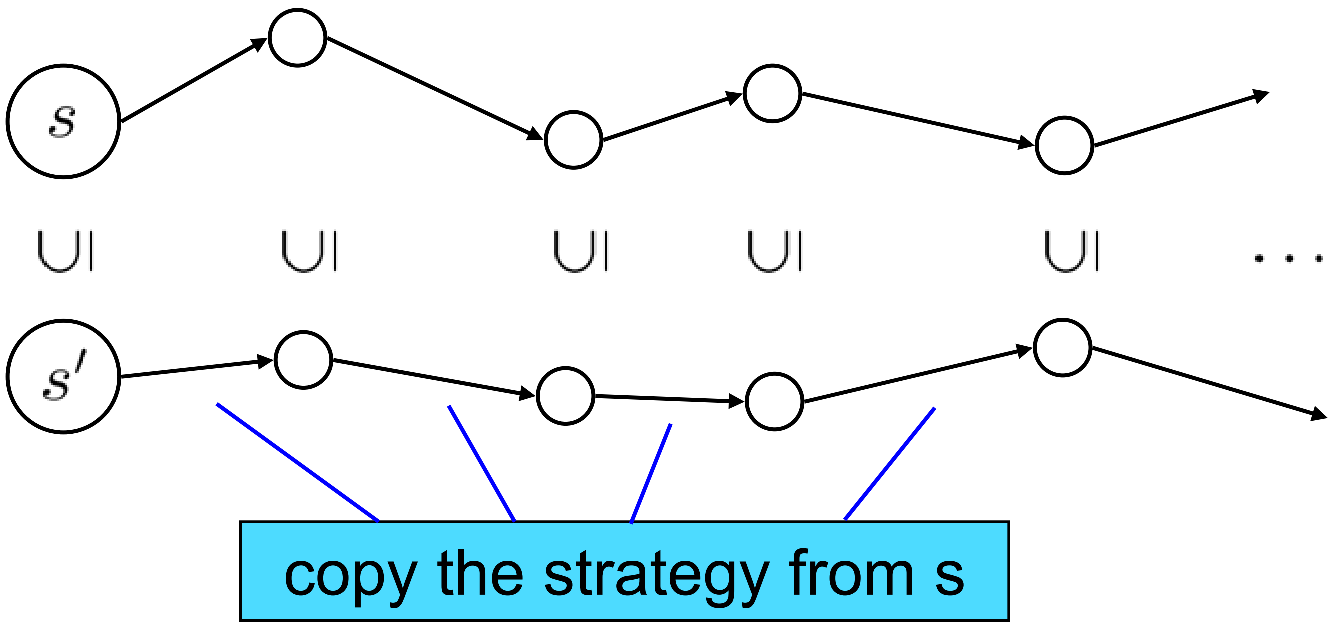
Perfect information

Direct symbolic algorithm

Intuition: if s is winning, then s′ $\subseteq$ s is also winning.

➔ The set of winning cells is downward-closed.

Symbolic algorithm

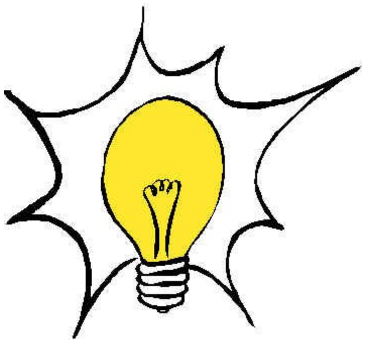


Intuition: if s is winning, then s′ $\subseteq$ s is also winning.

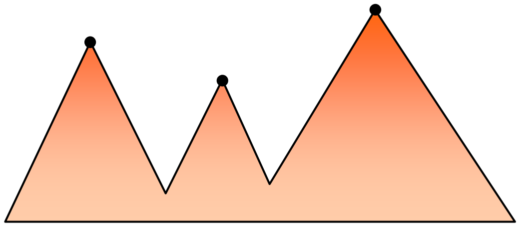
➡ The set of winning cells is downward-closed.

Antichains

- Winning knowledge-sets are downward-closed
- Useful operations preserve downward-closedness



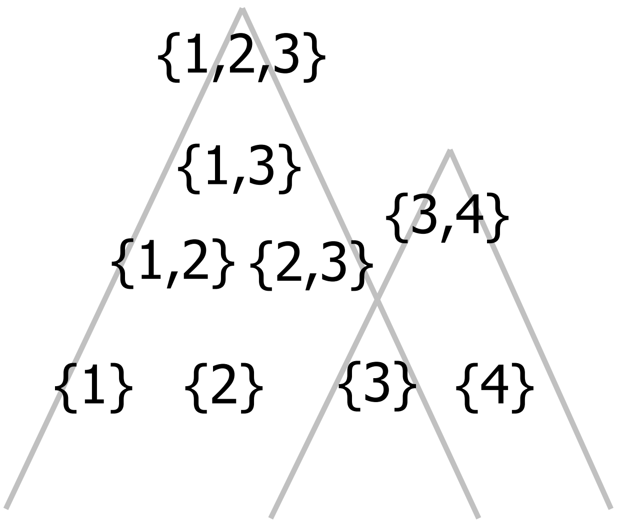
Compact representation using maximal elements $\rightarrow$ Antichains



Antichains

The antichain {{1,2,3},{3,4}}

represents the set of cells



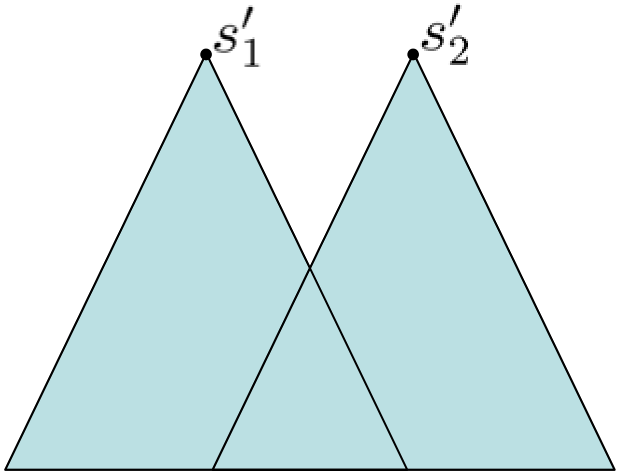
$= \downarrow\{\{1,2,3\},\{3,4\}\}$

i.e. the downward-closure of {{1,2,3},{3,4}}

Structure of antichains

Membership

$\cdot\, s$ $\overset{?}{\in}$

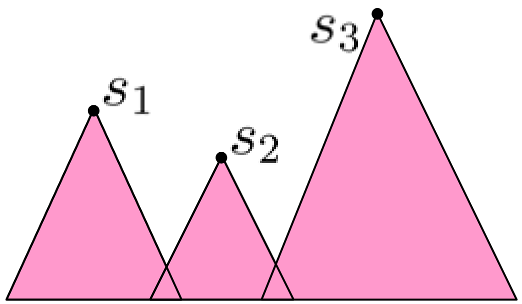


$q' = \{s'_1, s'_2\}$

$s \in \downarrow q'$ iff $\exists s' \in q' : s \subseteq s'$

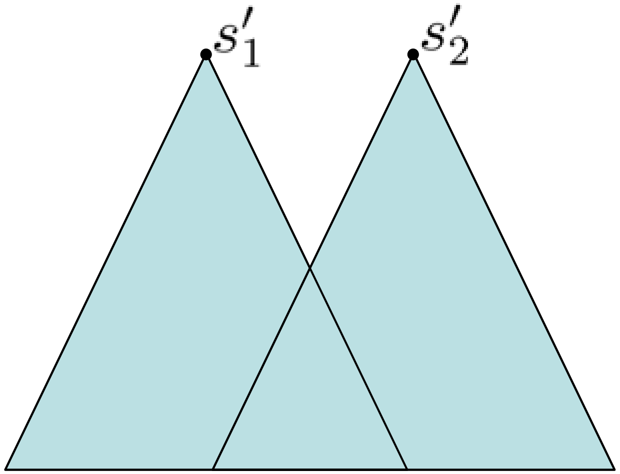
Structure of antichains

Inclusion



$$q = \{s_1, s_2, s_3\}$$

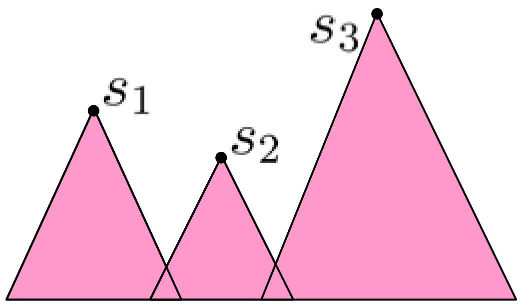
$$\overset{?}{\subseteq}$$



$$q' = \{s'_1, s'_2\}$$

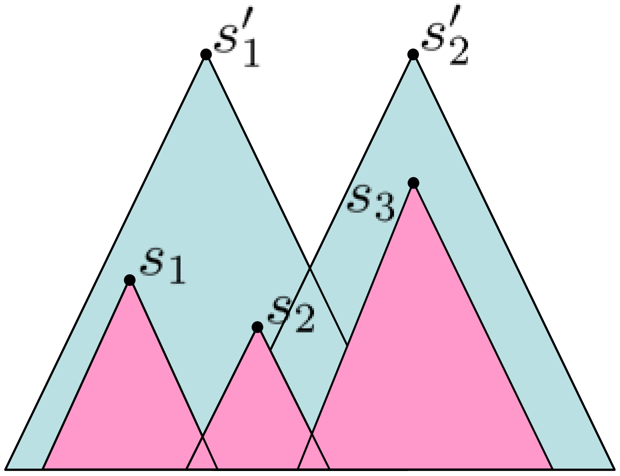
Structure of antichains

Inclusion



$q = \{s_1, s_2, s_3\}$

$\overset{?}{\subseteq}$



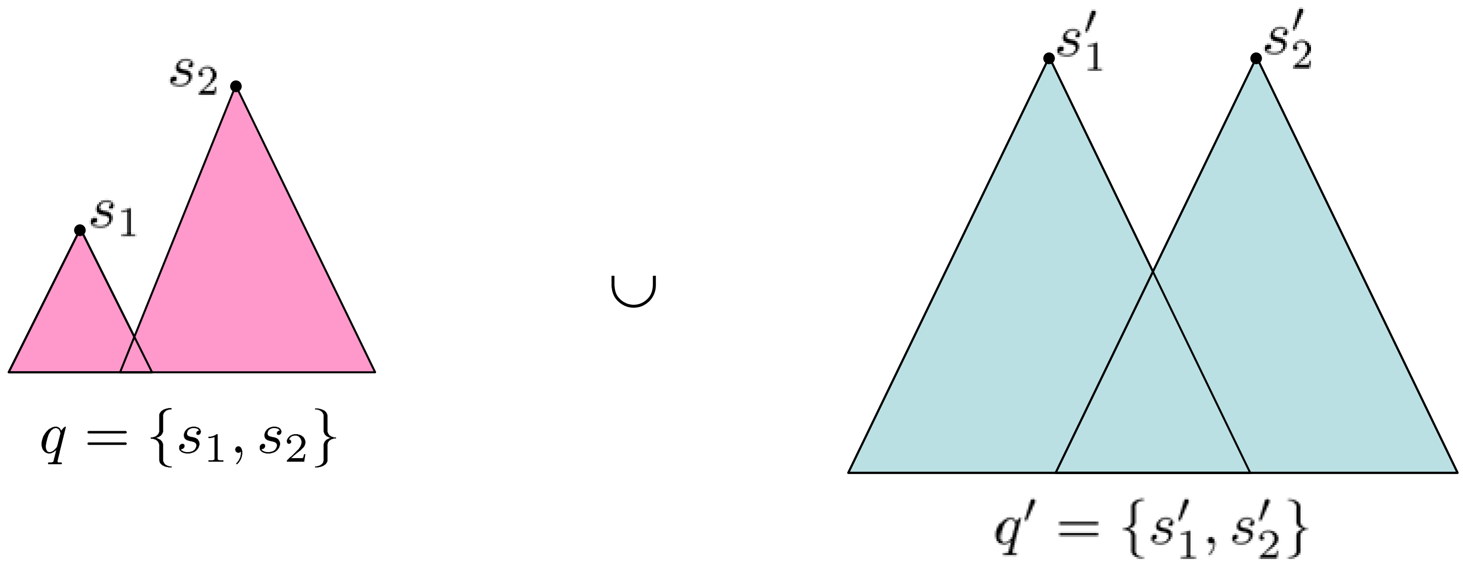
$q' = \{s'_1, s'_2\}$

$q \sqsubseteq q'$ iff $\forall s \in q \cdot \exists s' \in q' : s \subseteq s'$

$\sqsubseteq$ partial order on antichains

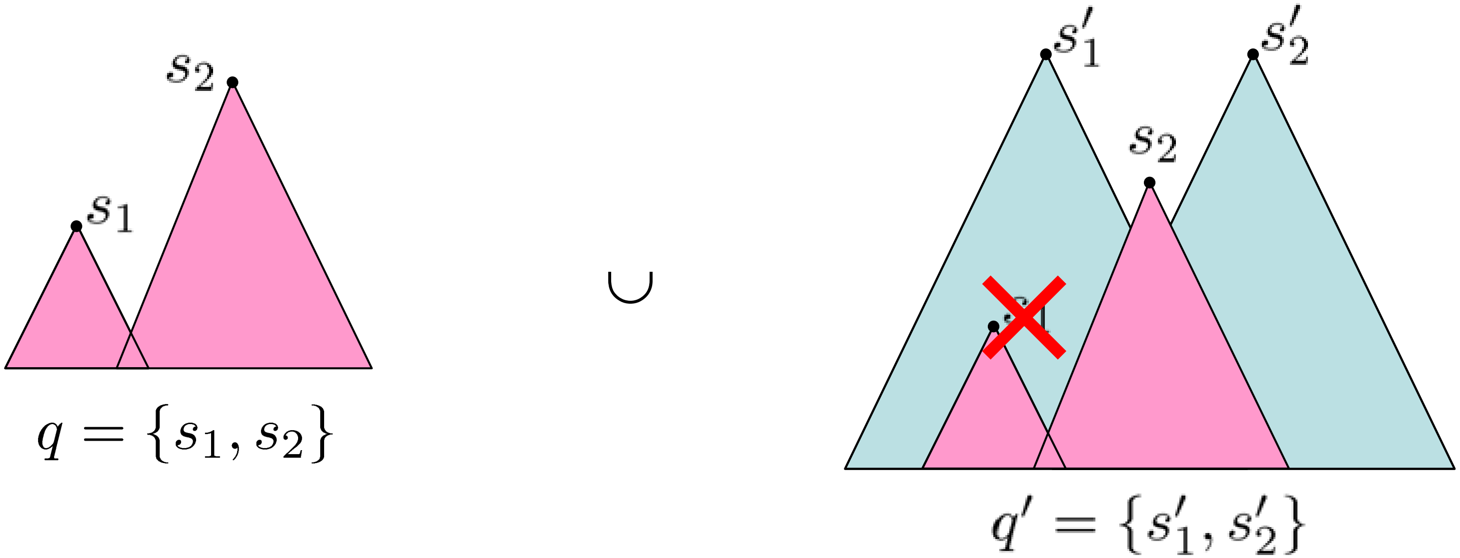
Structure of antichains

Union



Structure of antichains

Union

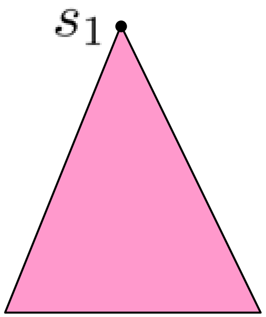


$q \sqcup q' =$ maximal elements of $q \cup q'$.

Computing $q_1 \sqcup q_2 \sqcup \ldots \sqcup q_n$ is polynomial.

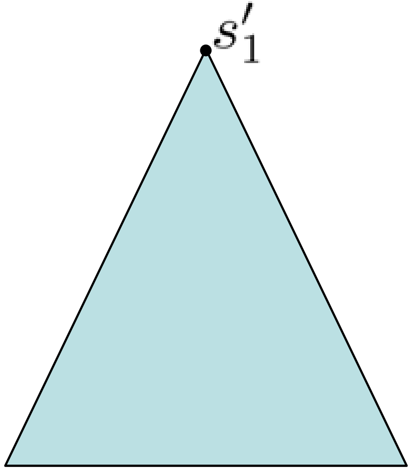
Structure of antichains

Intersection



$q = \{s_1\}$

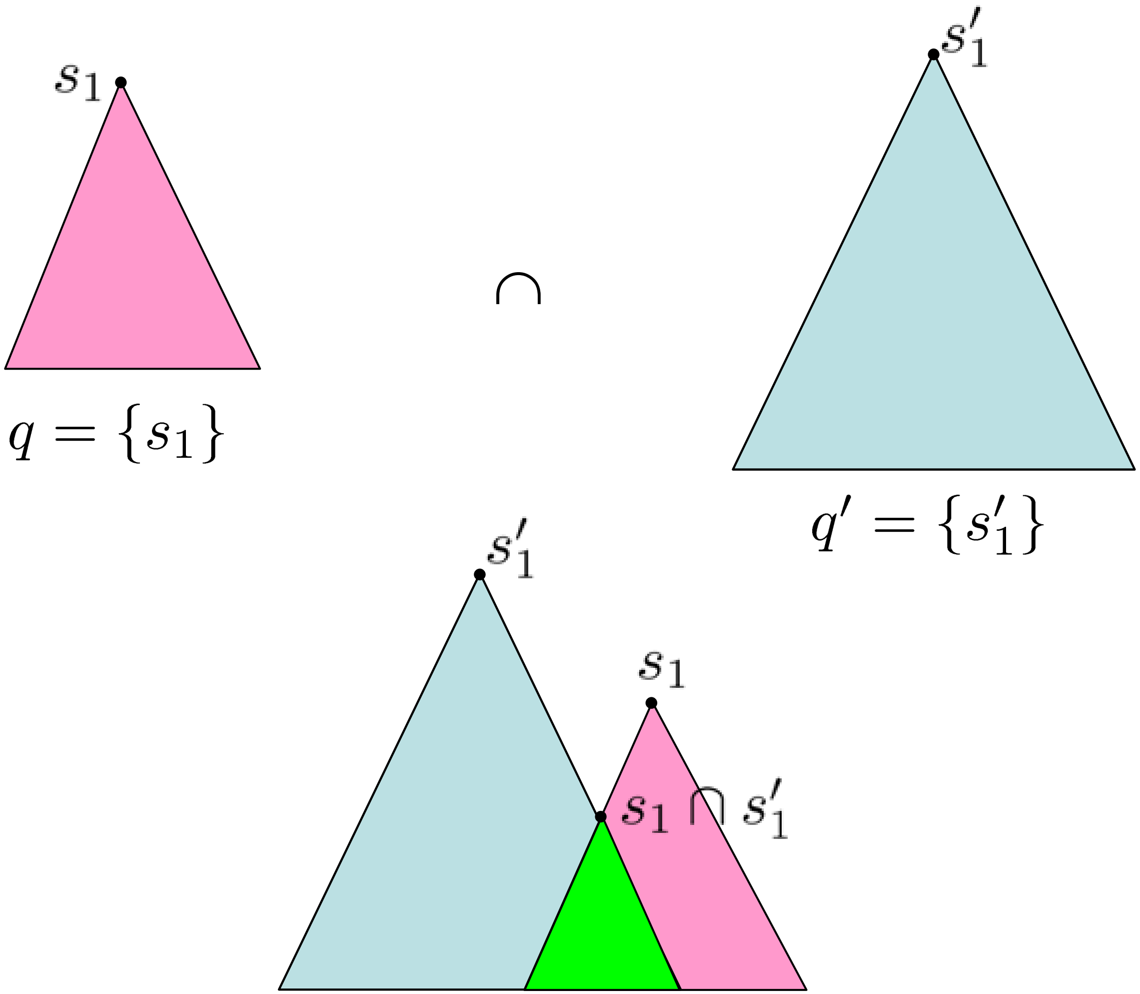
$\cap$



$q' = \{s'_1\}$

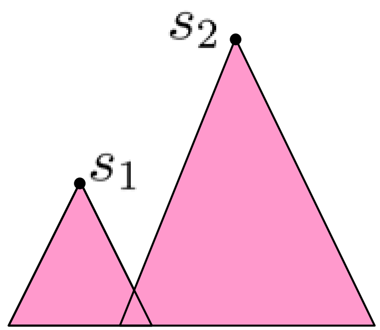
Structure of antichains

Intersection



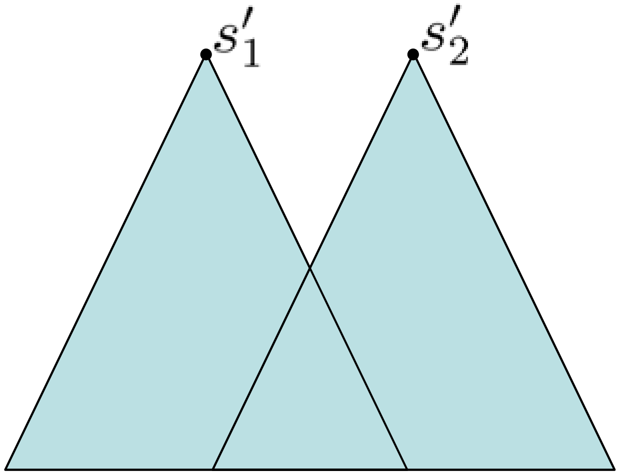
Structure of antichains

Intersection



$q = \{s_1, s_2\}$

$\cap$

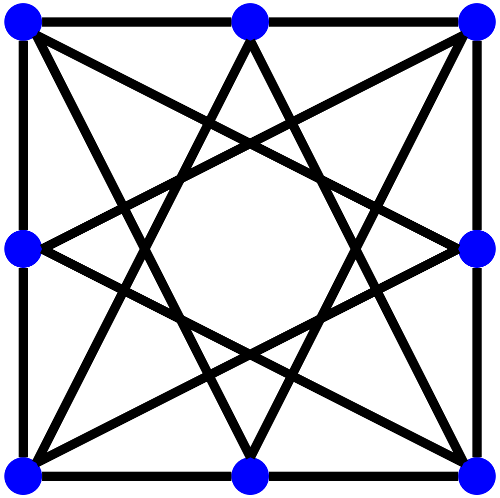


$q' = \{s'_1, s'_2\}$

$q \sqcap q' =$ maximal elements of $\{s \cap s' \mid s \in q \wedge s' \in q'\}$.

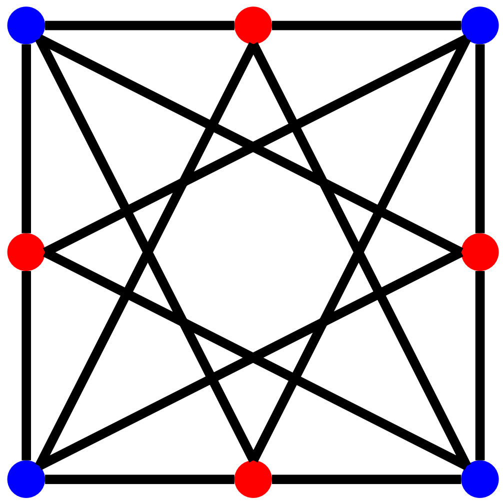
Computing $q_1 \sqcap q_2 \sqcap \ldots \sqcap q_n$ is exponential !

Structure of antichains



Independent set
(pairwise non-adjacent vertices)

Structure of antichains



Independent set
(pairwise non-adjacent vertices)

Computing largest independent set is NP-hard

Structure of antichains

Consider a graph $G = (V, E)$

The sets of vertices that do no contain edge (v,w) are represented by the antichain $\{V \setminus \{v\}, V \setminus \{w\}\}$

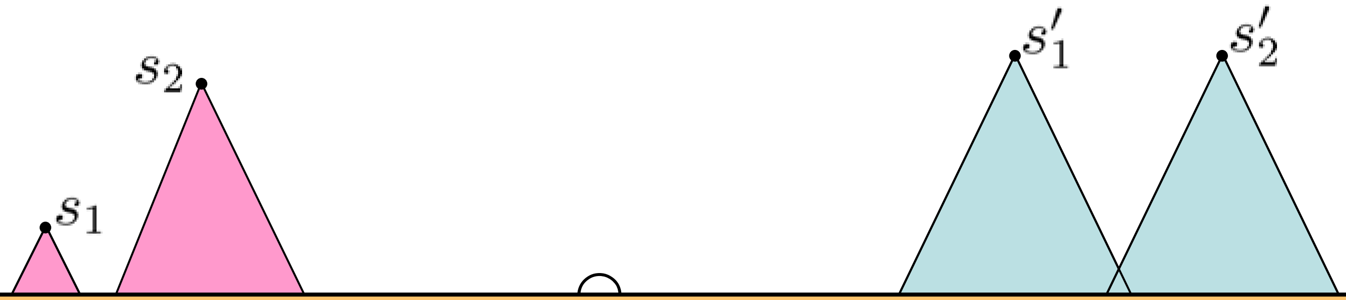
Hence, the maximal independent sets of G are defined by

$$\bigsqcap_{(v,w)\in E} \{V \setminus \{v\}, V \setminus \{w\}\}$$

Computing $q_1 \sqcap q_2 \sqcap \ldots \sqcap q_n$ is exponential (unless P=NP)

Structure of antichains

Intersection



Antichains partially-ordered by $\sqsubseteq$ is
a complete lattice

$q \sqcap q' =$ maximal elements of $\{s \cap s' \mid s \in q \wedge s' \in q'\}$.

Computing $q_1 \sqcap q_2 \sqcap \ldots \sqcap q_n$ is exponential !

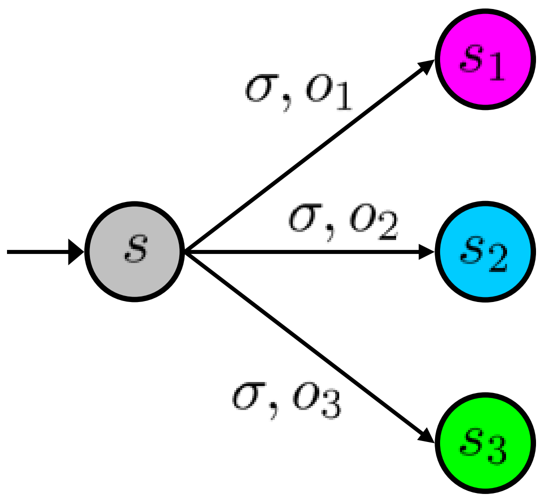
Symbolic algorithm

Controllable predecessor operator

CPre(Y) = cells s from which Player 1 has an action (σ)

such that for all obs chosen by Player 2

the cell $\text{post}_\sigma(s) \cap obs$ is in Y



$\text{post}_\sigma(s) \cap o_1$ in Y

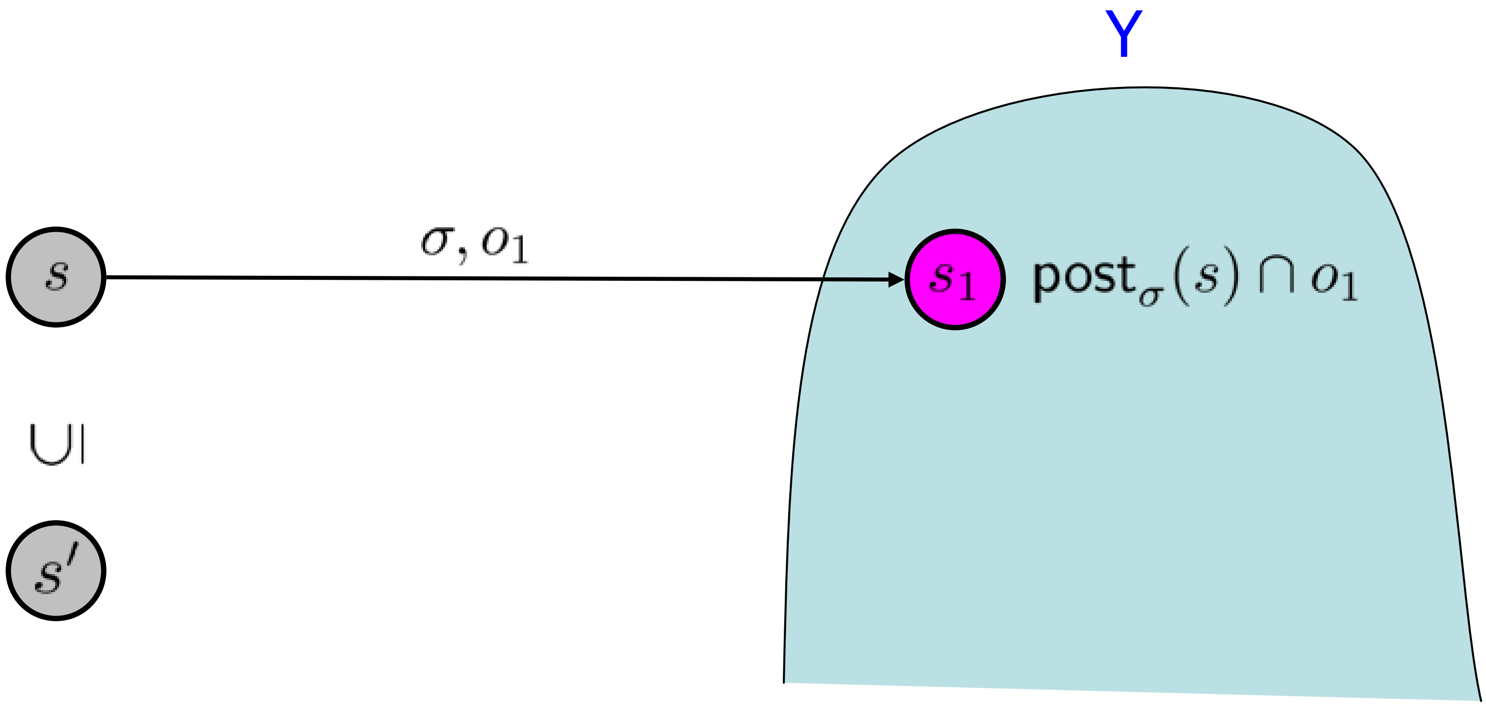
$\text{post}_\sigma(s) \cap o_2$ in Y

$\text{post}_\sigma(s) \cap o_3$ in Y

Symbolic algorithm

Controllable predecessor operator

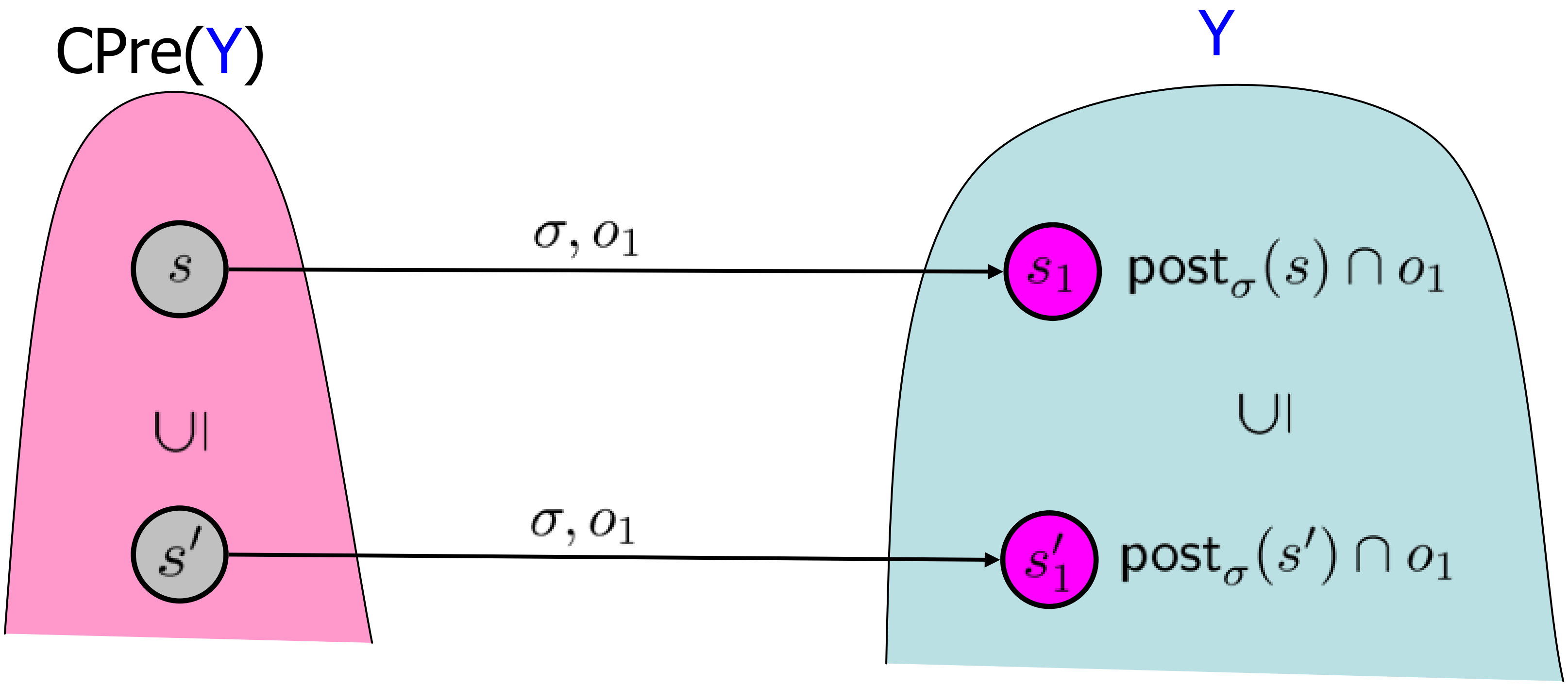
If Y is downward-closed, then ...



Symbolic algorithm

Controllable predecessor operator

If Y is downward-closed, then CPre(Y) is downward-closed.



Cpre() preserves downward-closedness.

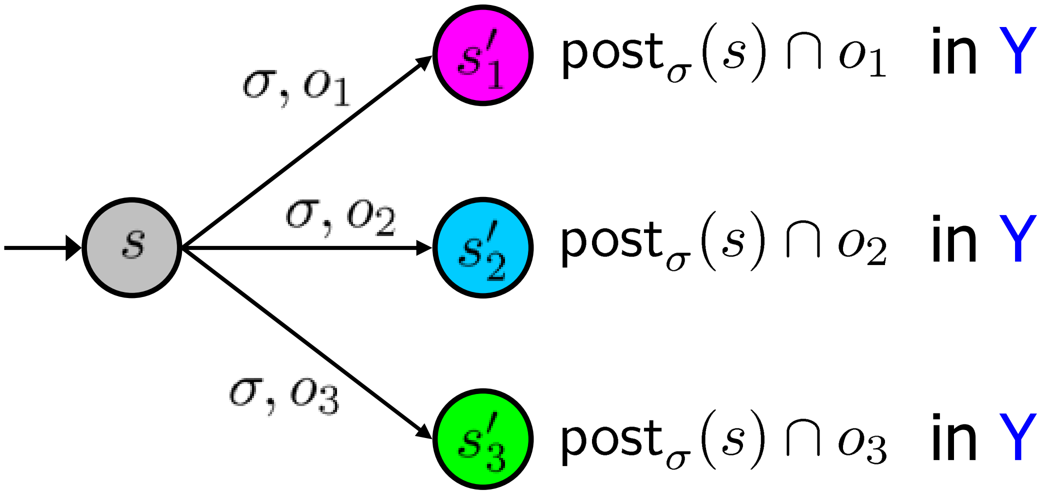
Symbolic algorithm

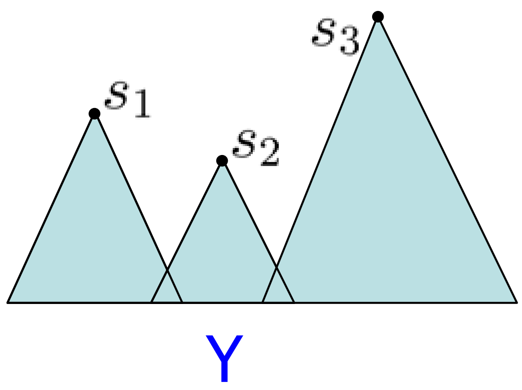
Controllable predecessor operator

CPre(Y) = cells s from which Player 1 has an action (σ)

such that for all obs chosen by Player 2

the cell $\text{post}_\sigma(s) \cap obs$ is in Y





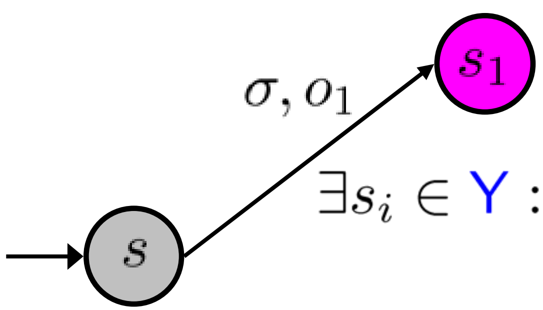
Symbolic algorithm

Controllable predecessor operator

CPre(Y) = cells s from which Player 1 has an action (σ)

such that for all obs chosen by Player 2

the cell $\mathrm{post}_\sigma(s) \cap obs$ is in Y

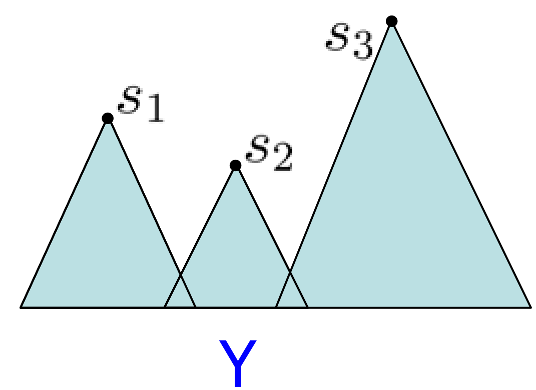


$\mathrm{post}_\sigma(s) \cap o_1$ in Y

$\exists s_i \in$ Y : $\mathrm{post}_\sigma(s) \cap o_1 \subseteq s_i$

$\mathrm{post}_\sigma(s) \subseteq s_i \cup \bar{o}_1$

$s \subseteq \widetilde{\mathrm{pre}}_\sigma(s_i \cup \bar{o}_1)$



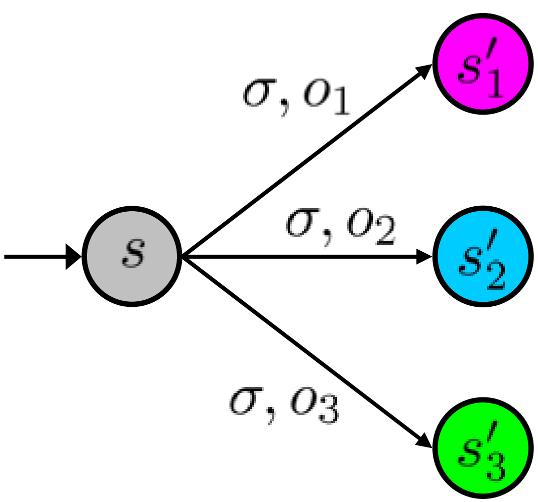
Symbolic algorithm

Controllable predecessor operator

CPre(Y) = cells s from which Player 1 has an action (σ)

such that for all obs chosen by Player 2

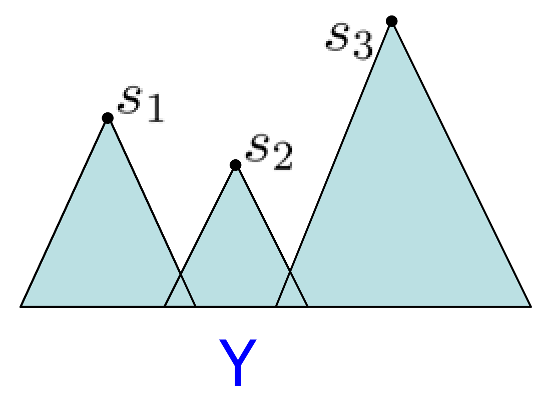
the cell $\mathrm{post}_\sigma(s) \cap obs$ is in Y



$\mathrm{post}_\sigma(s) \cap o_1 \subseteq s_i$

$\mathrm{post}_\sigma(s) \cap o_2 \subseteq s_j$

$\mathrm{post}_\sigma(s) \cap o_3 \subseteq s_k$



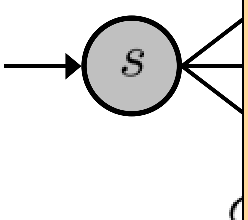
Symbolic algorithm

Controllable predecessor operator

CPre(Y) = cells s from which Player 1 has an action (σ)

such that for all obs chosen by Player 2

the cell $\mathsf{post}_\sigma(s) \cap obs$ is in Y



- combinatorially hard to compute

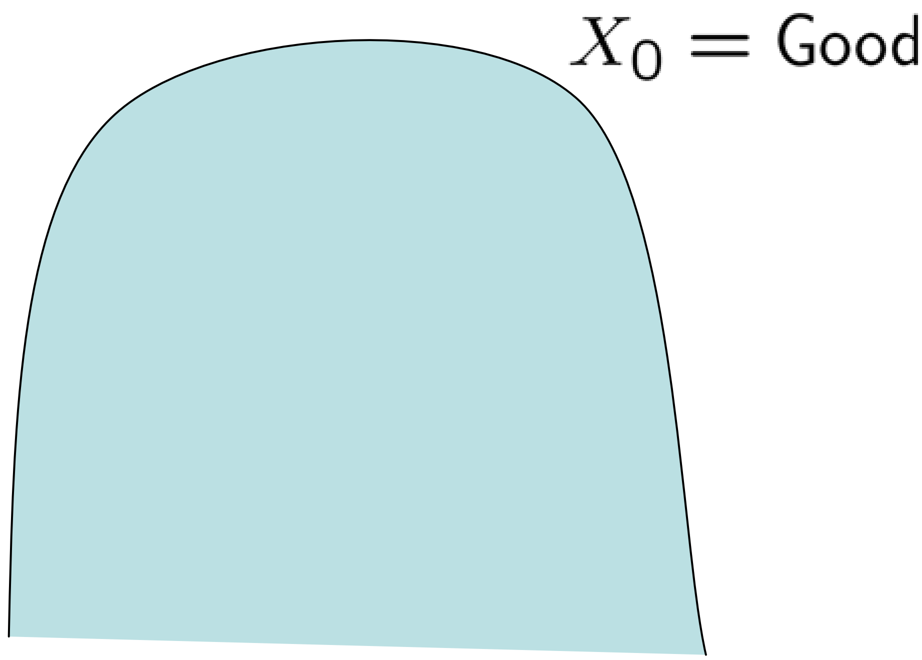
$$\mathsf{CPre}(Y) = \bigsqcup_{\sigma \in \Sigma} \bigsqcap_{o \in \mathsf{Obs}} \bigsqcup_{s' \in Y} \{\widetilde{\mathsf{pre}}_\sigma(s' \cup \bar{o})\}$$

- implemented using BDDs

Symbolic algorithm

Safety game: avoid Bad

$\text{Good} = \{s \mid s \cap \text{Bad} = \varnothing\}$ Good is downward-closed !

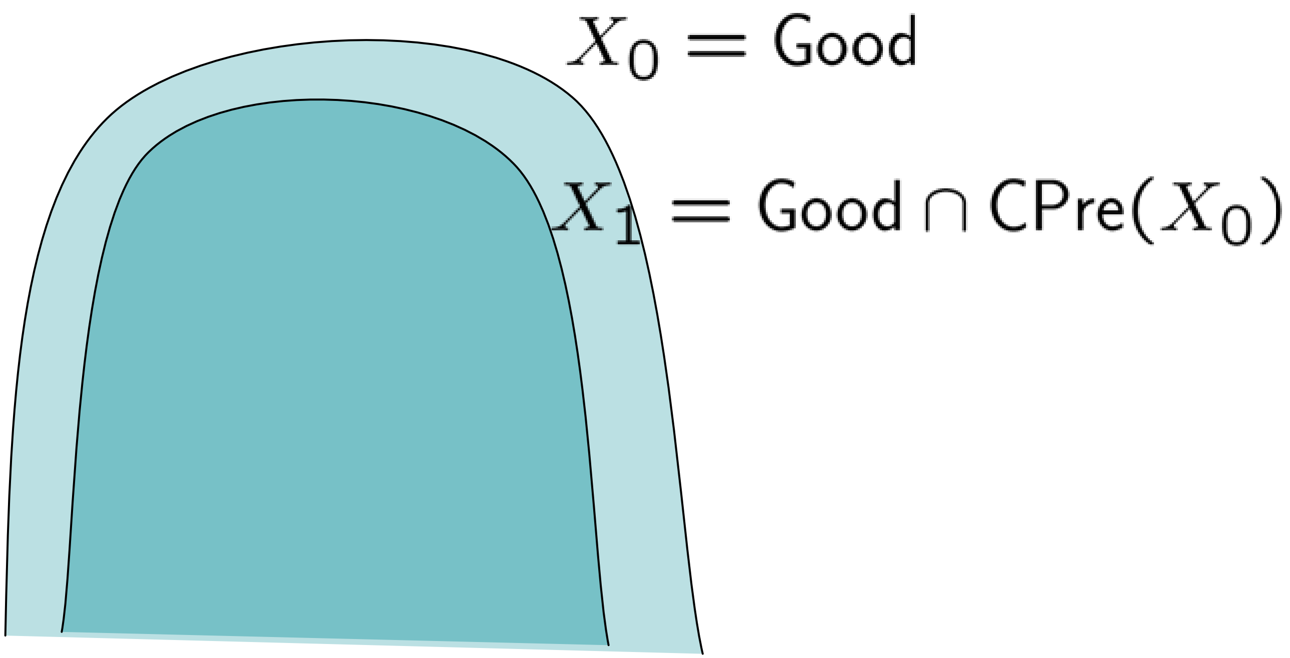


Symbolic algorithm

Safety game: avoid Bad

$\mathsf{Good} = \{s \mid s \cap \mathsf{Bad} = \varnothing\}$ Good is downward-closed !

cells winning in 1 step: Good $\cap$ CPre(Good)

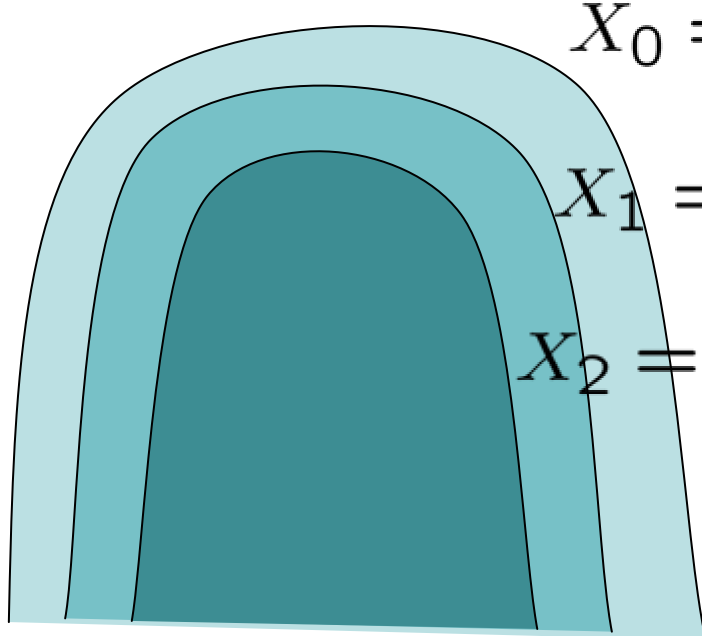


Symbolic algorithm

Safety game: avoid Bad

$\text{Good} = \{ s \mid s \cap \text{Bad} = \varnothing \}$ Good is downward-closed !

cells winning in 2 steps: Good $\cap$ CPre(Good) $\cap$ CPre(X_1)



$X_0 = \text{Good}$

$X_1 = \text{Good} \cap \text{CPre}(X_0)$

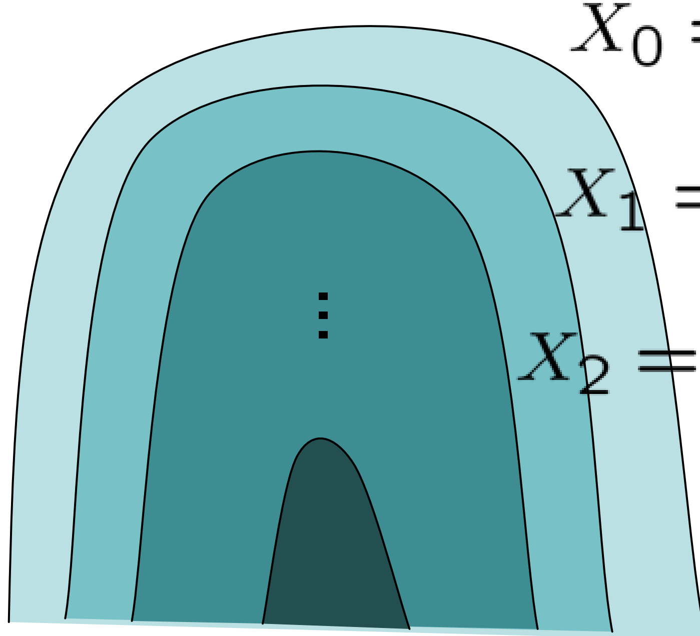
$X_2 = \text{Good} \cap \text{CPre}(X_1)$

Symbolic algorithm

Safety game: avoid Bad

$\mathsf{Good} = \{ s \mid s \cap \mathsf{Bad} = \varnothing \}$ Good is downward-closed !

cells winning in k steps: $\nu X \cdot \mathsf{Good} \cap \mathsf{CPre}(X)$



$X_0 = \mathsf{Good}$

$X_1 = \mathsf{Good} \cap \mathsf{CPre}(X_0)$

$X_2 = \mathsf{Good} \cap \mathsf{CPre}(X_1)$

$\vdots$

X_k

Symbolic algorithm

Safety game: avoid Bad

$\mathsf{Good} = \{s \mid s \cap \mathsf{Bad} = \varnothing\}$ Good is downward-closed !

cells winning in k steps: $\nu X \cdot \mathsf{Good} \cap \mathsf{CPre}(X)$

Fixpoint after at most $O(2^n)$ iterations

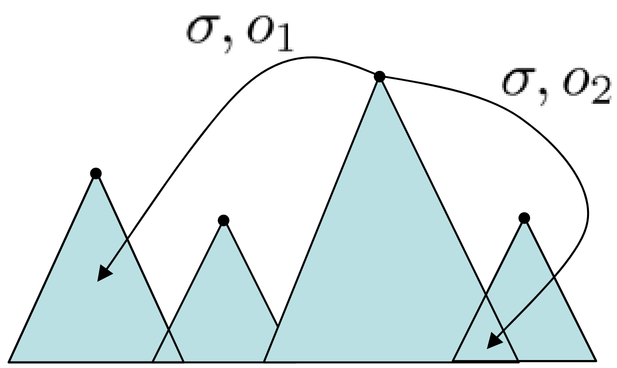
Computing CPre() is $O(2^{|obs|})$

…but exponentially more succinct sets !

Strategy construction

Safety game: avoid Bad

From every winning cell, Player 1 has an action to stay in the set of winning cells

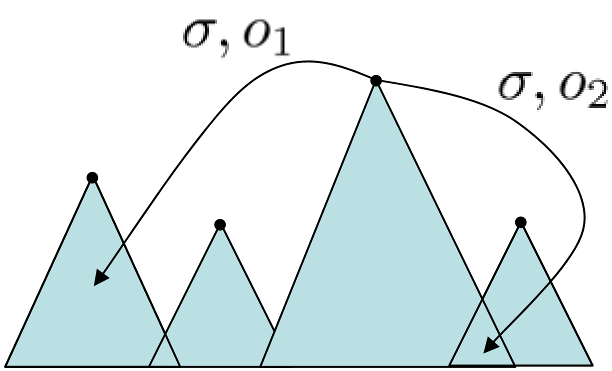


Winning cells

Strategy construction

Safety game: avoid Bad

From every winning cell, Player 1 has an action to stay in the set of winning cells



Winning cells

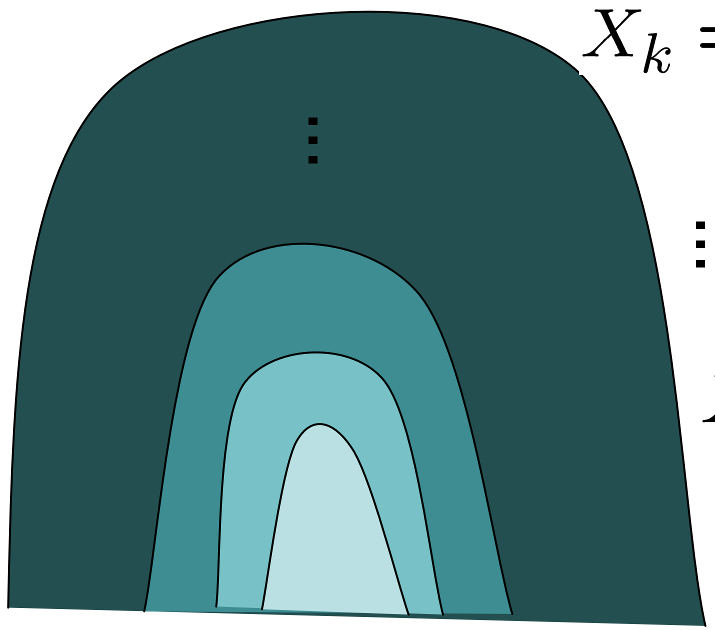
Strategy automaton

(Moore machine)

Symbolic algorithm

Reachability game: reach Target

cells winning in k steps: $\mu X \cdot \text{Target} \cup \text{CPre}(X)$



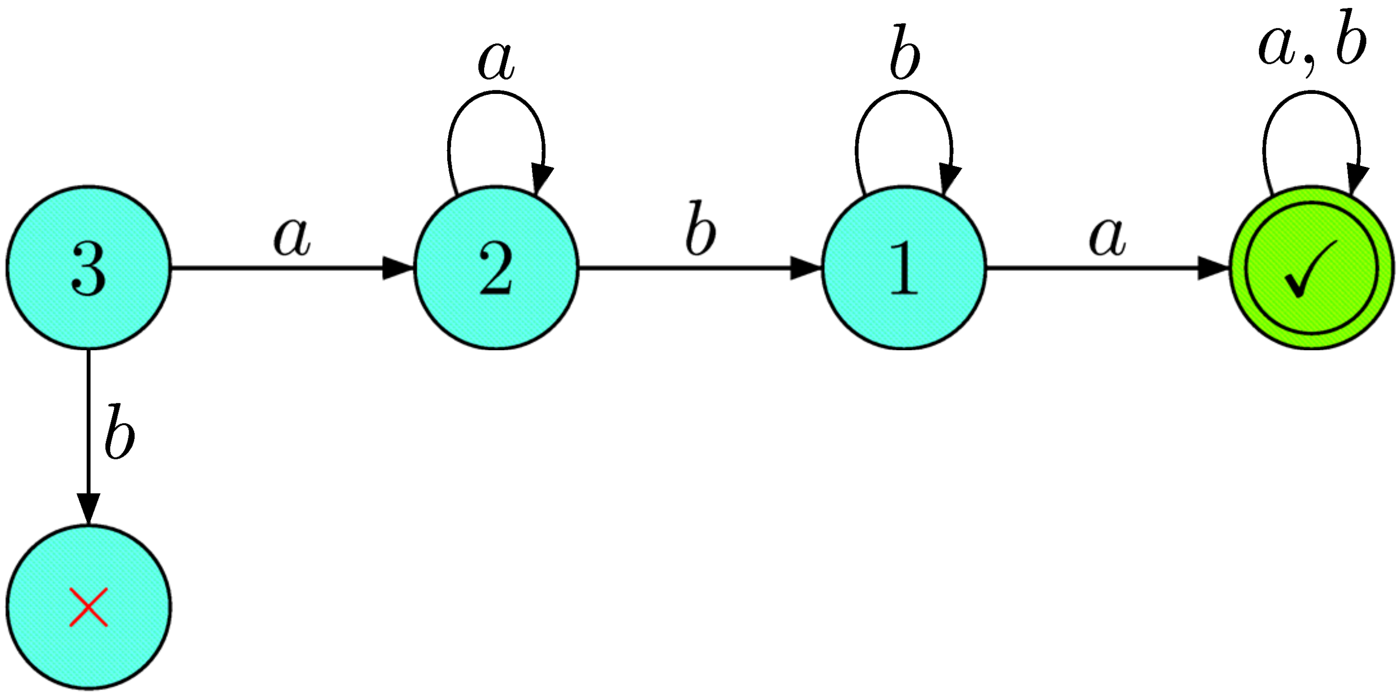
$X_k = \text{Target} \cup \text{CPre}(X_{k-1})$

⋮

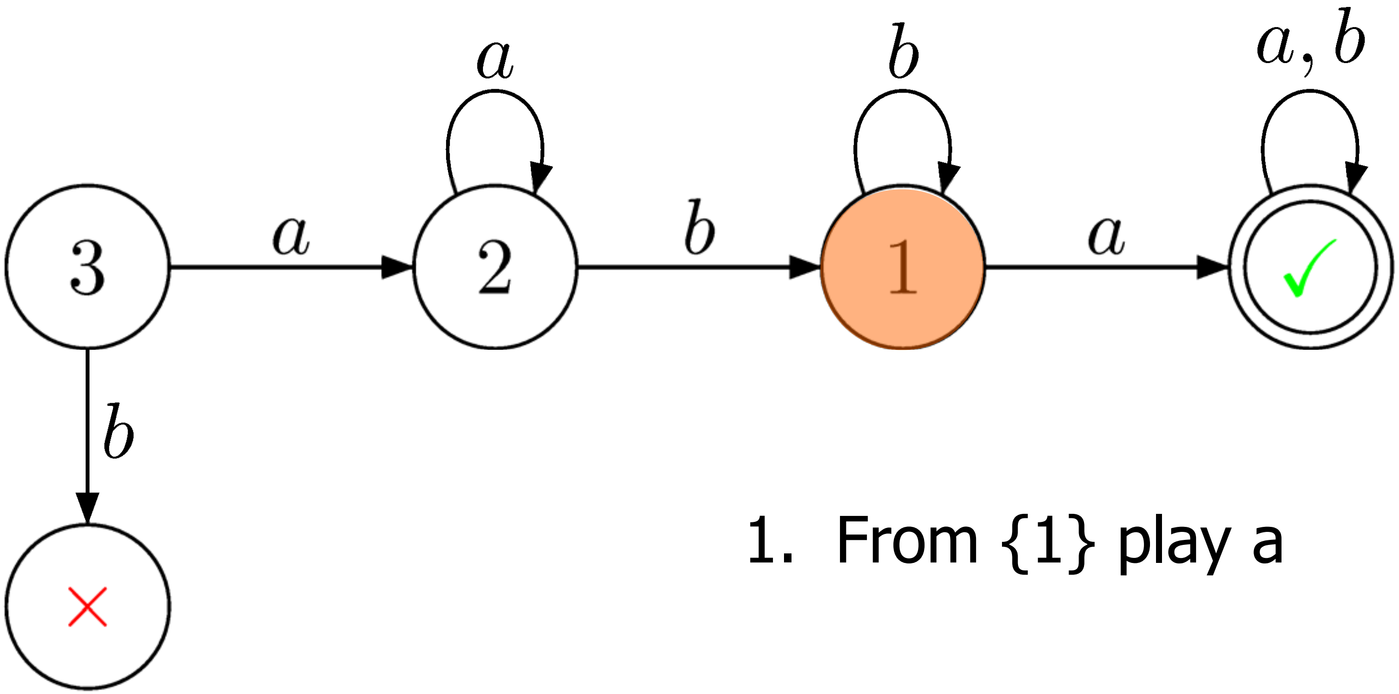
$X_1 = \text{Target} \cup \text{CPre}(X_0)$

$X_0 = \text{Target}$

Reachability

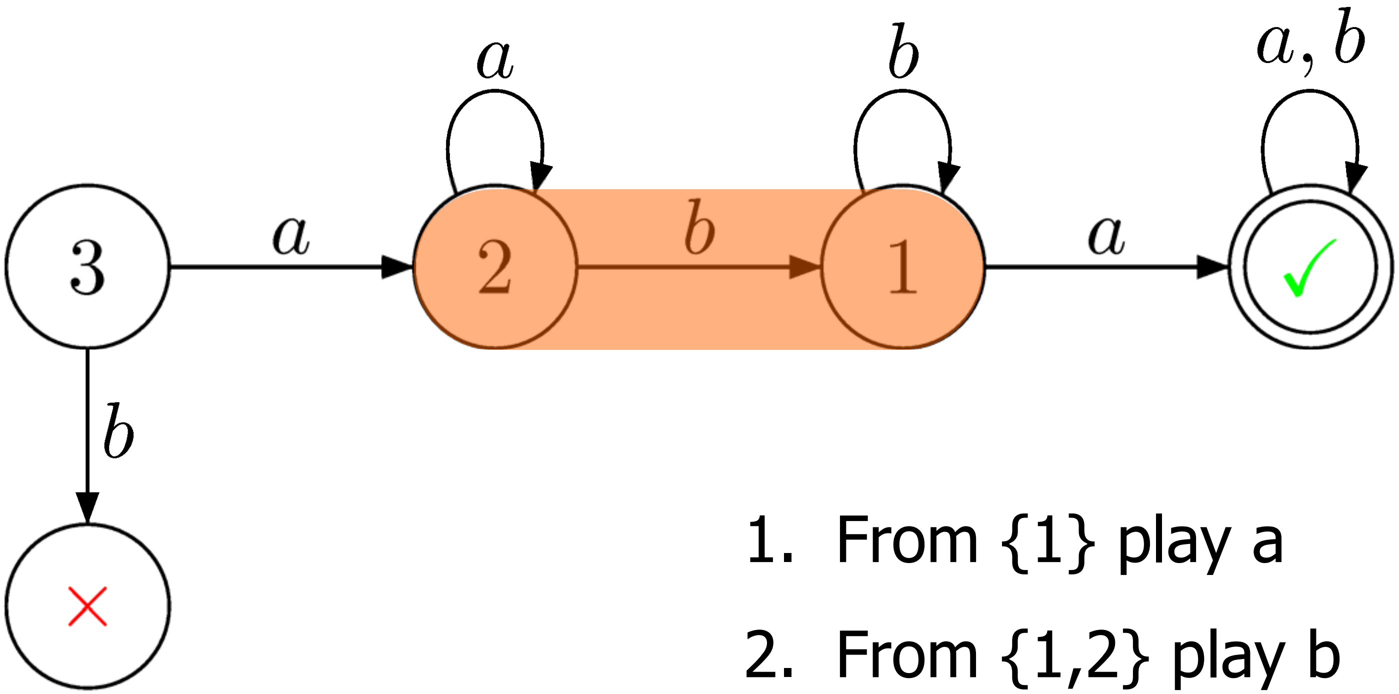


Reachability



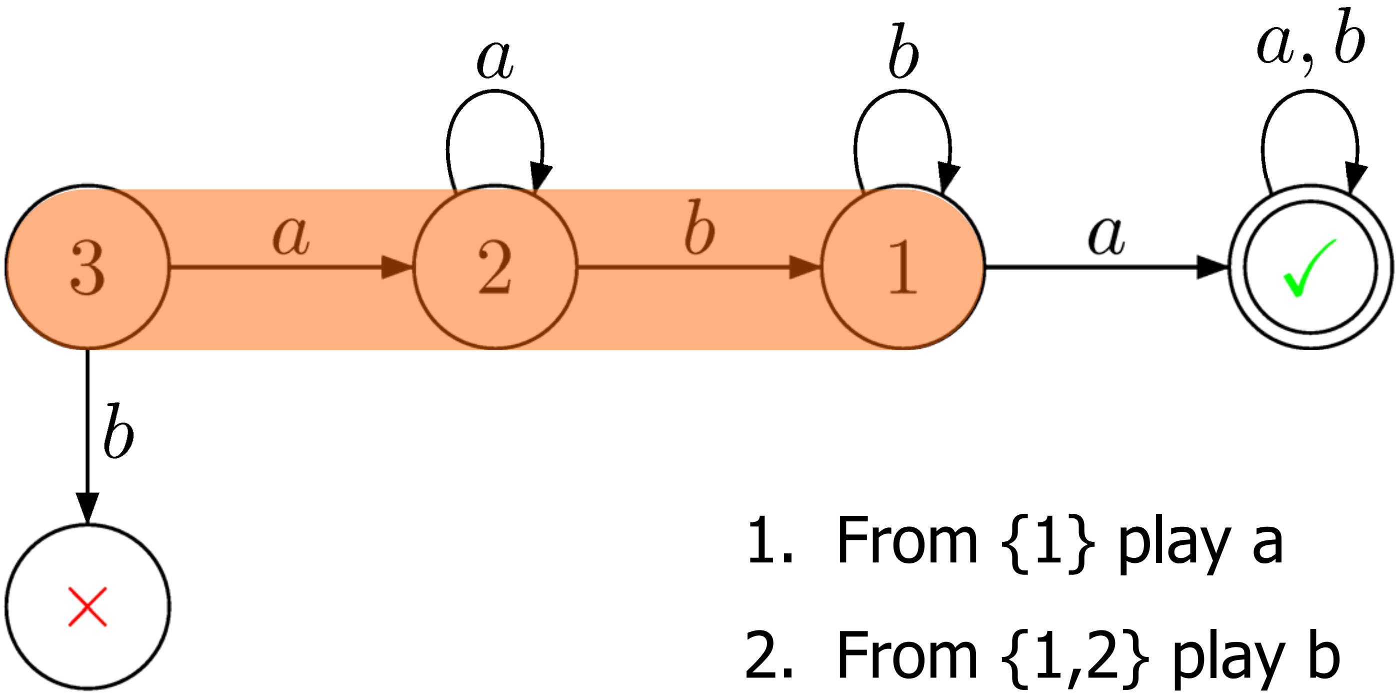
1. From {1} play a

Reachability



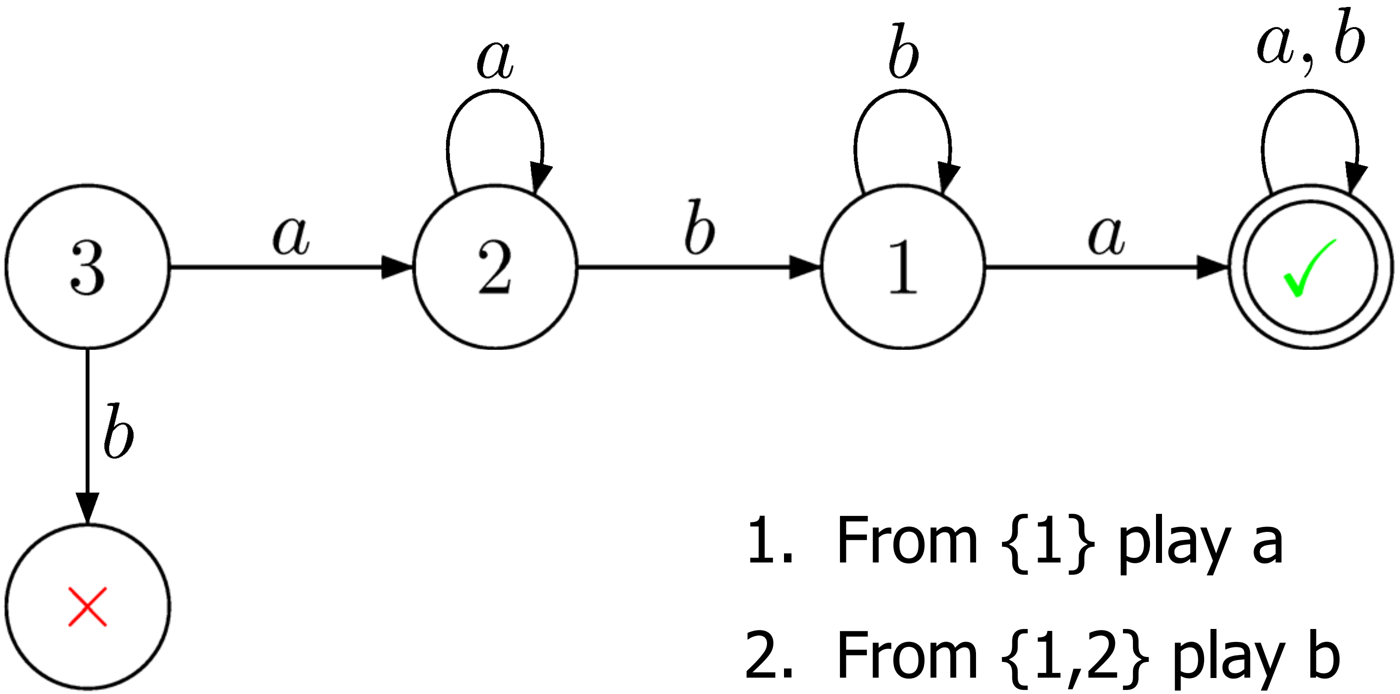
1. From {1} play a
2. From {1,2} play b

Reachability



1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Reachability

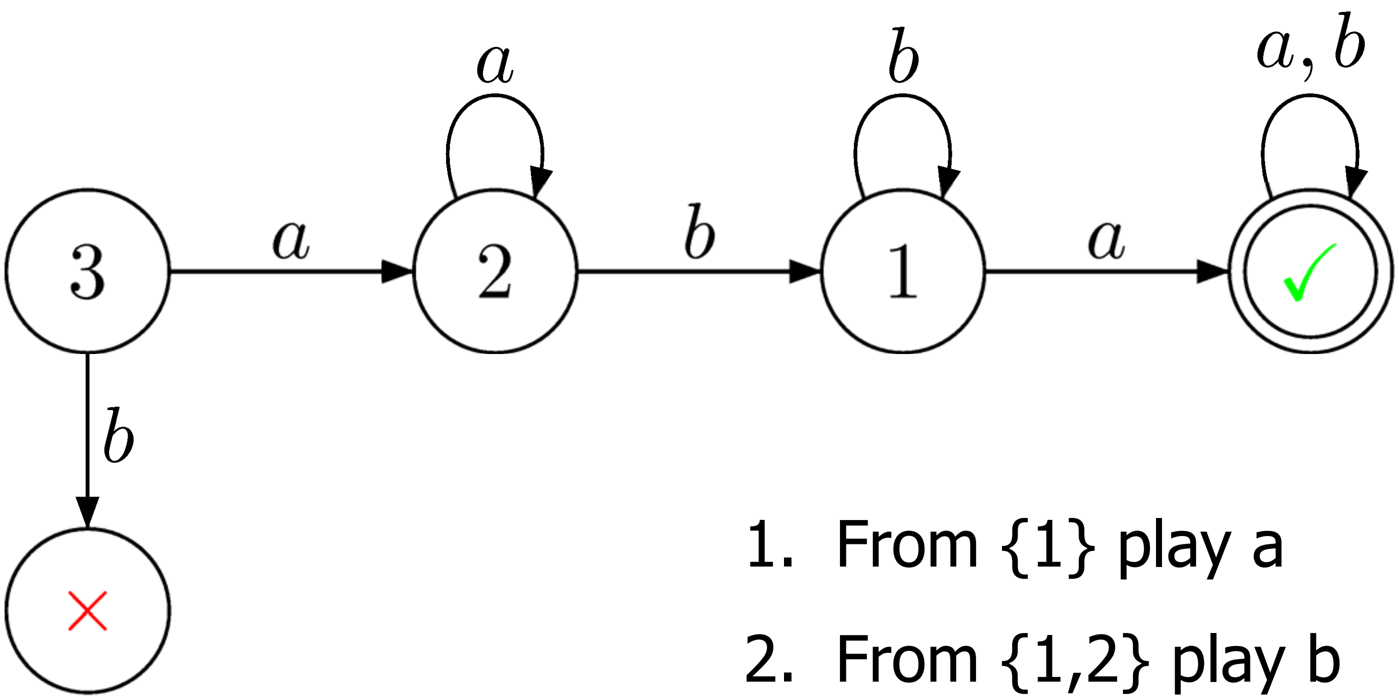


1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Fixpoint of winning cells: {{1,2,3}}

Winning strategy ??

Reachability

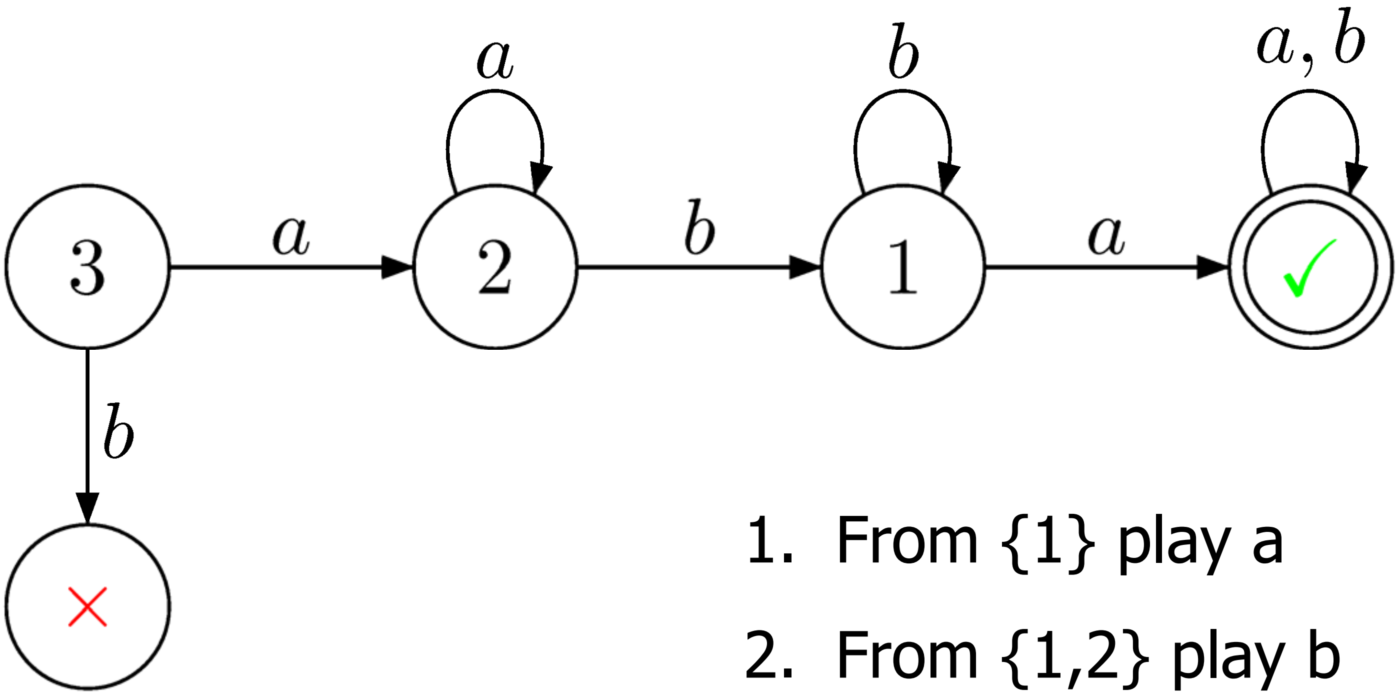


1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Fixpoint of winning (cell, action): $\{\{1,2,3\}_a,\{1,2\}_b\}$

Winning strategy ??

Reachability



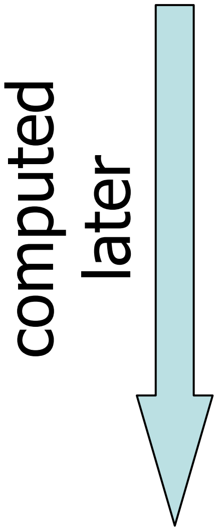
1. From {1} play a
2. From {1,2} play b
3. From {1,2,3} play a

Winning strategy

Current knowledge K: select earliest (cell,action)

such that K $\subseteq$ cell, play action

Strategy simplification #1



1. From {1,2,3} play a
2. From … play …
3. From … play …
4. From … play …
5. From {2} play b

Strategy simplification #1

computed later

1. From {1,2,3} play a
2. From … play …
3. From … play …
4. From … play …
5. ~~From {2} play b~~ Not necessary !

Rule 1: delete subsumed pairs computed later

Strategy simplification #2

computed later

1. From {1,2} play a
2. From {3,4} play …
3. From {1,3} play a
4. From {3,5} play …
5. From {1,2,3} play a

Strategy simplification #2

computed later

1. ~~From {1,2} play a~~ Not necessary !
2. From {3,4} play …
3. From {1,3} play a
4. From {3,5} play …
5. From {1,2,3} play a

Rule 2: delete strongly-subsumed pairs

Alpaga

First prototype for solving parity games of imperfect information

- Use antichains as compact representation of winning sets of positions
- Compute Controllable Predecessor with BDDs
- Publish Reachability/Safety attractor moves to compose the strategy (earlier published move sticks)
- Strategy simplification

Alpaga

First prototype for solving parity games of imperfect information

- Implemented in Python + CUDD
- ≤1000 LoC
- Solves 50 states, 28 observations, 3 priorities (explicit game graph)

http://www.antichains.be/alpaga

Some experiments

	Size	Obs	Priorities	Time (s)
Game1	4	4	Reach.	.1
Game2	3	2	Reach.	.1
Game3	6	3	3	.1
Game4	8	5	5	1.4
Game5	8	5	7	9.4
Game6	11	9	10	50.7
Game7	11	8	10	579.0
Locking	22	14	Safety	.6
Mutex	50	28	3	57.7

http://www.antichains.be/alpaga

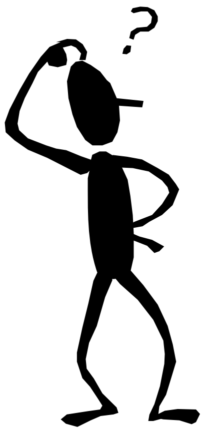
Alpaga

First prototype for solving parity games of imperfect information

Outlook

- Symbolic game graph
- Compact representation of strategies
- Almost-sure winning
- Relaxing visibility

Thank you !



Questions ?

Alpaga

http://www.antichains.be/alpaga

References

- [Reif84] J. H. Reif. The Complexity of Two-Player Games of Incomplete Information. J. Comput. Syst. Sci. 29(2): 274-301, 1984
- [CSL’06] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for Omega-regular Games of Incomplete Information. Proc. of CSL, LNCS 4207, Springer, 2006, pp. 287-302
- [Concur’08] D. Berwanger, K. Chatterjee, L. Doyen, T. A. Henzinger, and S. Raje. Strategy Construction for Parity Games with Imperfect Information. Proc. of Concur, LNCS 5201, Springer, 2008, pp. 325-339